19

Board Support Packages

19.1 Introduction

A **board support package** (**BSP**) is a collection of user-provided facilities which interface **RTEMS** and an application with a specific hardware platform. These facilities may include hardware initialization, device drivers, user extensions, and a **Multiprocessor Communications Interface** (**MPCI**). However, a minimal BSP need only support processor reset and initialization and, if needed, a clock tick.

19.2 Reset and Initialization

An **RTEMS** based application is initiated or re-initiated when the processor is reset. This initialization code is responsible for preparing the target platform for the **RTEMS** application. Although the exact actions performed by the initialization code are highly processor and target dependent, the logical functionality of these actions are similar across a variety of processors and target platforms.

Normally, the application's initialization is performed at two separate times: before the call to **initialize_executive** (reset application initialization) and after **initialize_executive** in the user's initialization tasks (local and global application initialization). The order of the startup procedure is as follows:

1. *Reset application initialization.*
2. *Call to* ***initialize_executive***

3. *Local and global application initialization.*

The reset application initialization code is executed first when the processor is reset. All of the hardware must be initialized to a quiescent state by this software before initializing **RTEMS**. When in quiescent state, devices do not generate any interrupts or require any servicing by the application. Some of the hardware components may be initialized in this code as well as any application initialization that does not involve calls to **RTEMS** directives.

The processor's Interrupt Vector Table which will be used by the application must be set to the required value by the reset application initialization code. Because interrupts are enabled automatically by **RTEMS** as part of the **initialize_executive** directive, the Interrupt Vector Table **MUST** be set before this directive is invoked to insure correct interrupt vectoring. The processor's Interrupt Vector Table must be accessible by **RTEMS** as it will be modified by the **interrupt_catch** directive. The reset code which is executed before the call to **initialize_executive** has the following requirements:

- *Must not make any **RTEMS** directive calls.*
- *If the processor supports multiple privilege levels, must leave the processor in the most privileged, or supervisory, state.*
- *Must allocate a stack of at least **MINIMUM_STACK_SIZE** bytes and initialize the stack pointer for the **initialize_executive** directive.*
- *Must initialize the processor's Interrupt Vector Table.*
- *Must disable all maskable interrupts.*
- *If the processor supports a separate interrupt stack, must allocate the interrupt stack and initialize the interrupt stack pointer.*

The **initialize_executive** directive does not return to the initialization code, but causes the highest priority initialization task to begin execution. Initialization tasks are used to perform both local and global application initialization which is dependent on **RTEMS** facilities. The user initialization task facility is typically used to create the application's set of tasks.

19.2.1 Interrupt Stack Requirements

The worst-case stack usage by interrupt service routines must be taken into account when designing an application. If the processor supports interrupt nesting, the stack usage must include the deepest nest level. The worst-case stack usage must account for the following requirements:

- *Processor's interrupt stack frame*

- *Processor's subroutine call stack frame*
- ***RTEMS** system calls requiring up to **MINIMUM_STACK_SIZE** bytes*
- *Registers saved on stack*
- *Application subroutine calls*

The **RTEMS** constant **MINIMUM_STACK_SIZE** includes all subroutine call stack frames and registers saved on the stack by RTEMS.

19.2.2 Processors with a Separate Interrupt Stack

Some processors support a separate stack for interrupts. When an interrupt is vectored and the interrupt is not nested, the processor will automatically switch from the current stack to the interrupt stack. The size of this stack is based solely on the worst-case stack usage by interrupt service routines.

The dedicated interrupt stack for the entire application is supplied and initialized by the reset and initialization code of the user's board support package. Since all ISRs use this stack, the stack size must take into account the worst case stack usage by any combination of nested ISRs.

19.2.3 Processors without a Separate Interrupt Stack

Some processors do not support a separate stack for interrupts. In this case, every task's stack must include enough space to handle the task's worst-case stack usage as well as the worst-case interrupt stack usage. This is necessary because the worst-case interrupt nesting could occur while any task is executing.

19.3 Device Drivers

Device drivers consist of control software for special peripheral devices and provide a logical interface for the application developer. The **RTEMS** I/O manager provides directives which allow applications to access these device drivers in a consistent fashion. A Board Support Package may include device drivers to access the hardware on the target platform. These devices typically include serial and parallel ports, counter/timer peripherals, real-time clocks, disk interfaces, and network controllers.

For more information on device drivers, refer to the **I/O Manager** chapter.

19.3.1 Clock Tick Device Driver

Most **RTEMS** applications will include a clock tick device driver which invokes the **clock_tick** directive at regular intervals. The clock tick is necessary if the application is to utilize timeslicing, the clock manager, the timer manager, the rate monotonic manager, or the timeout option on blocking directives.

The clock tick is usually provided as an interrupt from a counter/timer or a real-time clock device. When a counter/timer is used to provide the clock tick, the device is typically programmed to operate in continuous mode. This mode selection causes the device to automatically reload the initial count and continue the countdown without programmer intervention. This reduces the overhead required to manipulate the counter/timer in the clock tick ISR and increases the accuracy of tick occurrences. The initial count can be based on the **microseconds_per_tick** field in the **RTEMS Configuration Table**. An alternate approach is to set the initial count for a fixed time period (such as one millisecond) and have the ISR invoke **clock_tick** on the **microseconds_per_tick** boundaries. Obviously, this can induce some error if the configured **microseconds_per_tick** is not evenly divisible by the chosen clock interrupt quantum.

It is important to note that the interval between clock ticks directly impacts the granularity of **RTEMS** timing operations. In addition, the frequency of clock ticks is an important factor in the overall level of system overhead. A high clock tick frequency results in less processor time being available for task execution due to the increased number of clock tick ISRs.

19.4 User Extensions

RTEMS allows the application developer to augment selected features by invoking user-supplied extension routines when the following system events occur:

- *Task creation*
- *Task initiation*
- *Task reinitiation*
- *Task deletion*
- *Task context switch*
- *Task exits*
- *Fatal error detection*

User extensions can be used to implement a wide variety of functions including execution profiling, non-standard coprocessor support, debug support, and error detection and recovery. For example, the context of a non-standard numeric coprocessor may be maintained via the user extensions. In this example, the task creation and deletion extensions are responsible for allocating and deallocating the

context area, the task initiation and reinitiation extensions would be responsible for priming the context area, and the task context switch extension would save and restore the context of the device.

For more information on user extensions, refer to the **User Extensions** chapter.

19.5 Multiprocessor Communications Interface (MPCI)

RTEMS requires that an MPCI layer be provided when a multiple node application is developed. This MPCI layer must provide an efficient and reliable communications mechanism between the multiple nodes. Tasks on different nodes communicate and synchronize with one another via the MPCI. Each MPCI layer must be tailored to support the architecture of the target platform.

For more information on the MPCI, refer to the **Multiprocessing Manager** chapter.

19.5.1 Tightly-Coupled Systems

A tightly-coupled system is a multiprocessor configuration in which the processors communicate solely via shared global memory. The MPCI can simply place the **RTEMS** packets in the shared memory space. The two primary considerations when designing an MPCI for a tightly-coupled system are data consistency and informing another node of a packet.

The data consistency problem may be solved using atomic "test and set" operations to provide a "lock" in the shared memory. It is important to minimize the length of time any particular processor locks a shared data structure.

The problem of informing another node of a packet can be addressed using one of two techniques. The first technique is to use an interprocessor interrupt capability to cause an interrupt on the receiving node. This technique requires that special support hardware be provided by either the processor itself or the target platform. The second technique is to have a node poll for arrival of packets. The drawback to this technique is the overhead associated with polling.

19.5.2 Loosely-Coupled Systems

A loosely-coupled system is a multiprocessor configuration in which the processors communicate via some type of communications link which is not shared global memory. The MPCI sends the **RTEMS** packets across the communications link to the destination node. The characteristics of the communications link vary widely and have a significant impact on the MPCI layer. For example, the bandwidth of the communications link has an obvious impact on the maximum MPCI throughput.

The characteristics of a shared network, such as Ethernet, lend themselves to supporting an MPCI layer. These networks provide both the point-to-point and broadcast capabilities which are expected by **RTEMS**.

19.5.3 Systems with Mixed Coupling

A mixed-coupling system is a multiprocessor configuration in which the processors communicate via both shared memory and communications links. A unique characteristic of mixed-coupling systems is that a node may not have access to all communication methods. There may be multiple shared memory areas and communication links. Therefore, one of the primary functions of the MPCI layer is to efficiently route **RTEMS** packets between nodes. This routing may be based on numerous algorithms. In addition, the router may provide alternate communications paths in the event of an overload or a partial failure.

19.5.4 Heterogeneous Systems

Designing an MPCI layer for a heterogeneous system requires special considerations by the developer. **RTEMS** is designed to eliminate many of the problems associated with sharing data in a heterogeneous environment. The MPCI layer need only address the representation of thirty-two (32) bit unsigned quantities.

For more information on supporting a heterogeneous system, refer the **Supporting Heterogeneous Environments** in the **Multiprocessing Manager** chapter.